

RASSUNTI PARTE B-C-D

Giacomo Minichi



MULTITHREADING

PROCESSO: entità del sistema operativo che rappresenta ciò che è necessario per eseguire un singolo programma.

contenuto da ↗ THREAD MULTIPLEX: flussi sequenziali di esecuzione, schedulabili individualmente.

RISORSE PROTEZIONE: dato della memoria principale, dato dell'I/O.

3 thread nello stesso spazio d'indirizzamento (COMPONENTE ATTIVA), tra di loro hanno una visione in un ambiente globale.

Ogni THREAD ha un THREAD CONTROL BLOCK (TCB), contiene informazioni su stato di esecuzione, info scheduling, info accounting, puntatori, eventuali altre informazioni.

Il ciclo di vita di un THREAD è composto dai seguenti stati: new: in fase di creazione

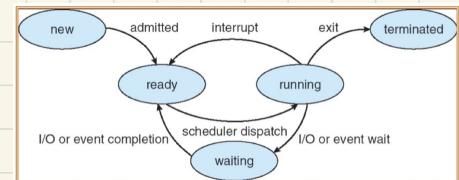
ready: in attesa, pronto per l'esecuzione

running: il processore esegue le sue istruzioni

waiting: in attesa di qualche evento

terminated: ha completato l'esecuzione

Quando un thread non è in esecuzione il suo TCB resta in coda d'attesa.



Il ciclo di ATTIVAZIONE (DISPATCHING) dei thread viene eseguito dal SO: 1) mettere in esecuzione il codice utente.
2) riprendere il controllo del thread in esecuzione.
3) scegliere il prossimo thread da eseguire.

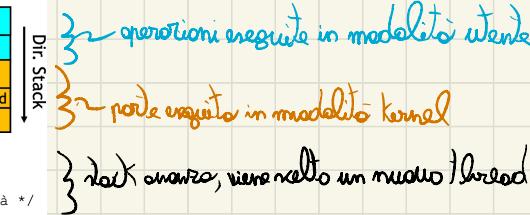
```
Loop {
    RunThread();
    ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(newTCB);
}
```

RUNTIME → so codice il suo thread nel PC
 " " codice l'ambiente
 relativa all'indirizzo presente nel PC

Per mantenere il controllo il pc utilizza

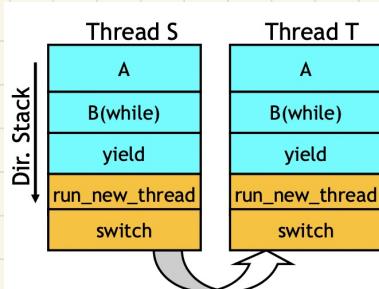
EVENTI INTERNI → blocco per effettuare I/O oppure evento di interruzione * } YIELD implicite
 effetto di un SEGNALE proveniente da un altro thread
 il thread esegue uno YIELD()

```
computePI() {
    while(TRUE) {
        ComputeNextDigit();
        yield();
    }
}
□ Per eseguire un nuovo thread:
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* altre attività */
}
```



* effetto di evento risveglio
 SOIN

* codice invoca una funzione
 il kernel invia l'operazione di read
 " " esegue la read



→ Quando si arriva allo switch del thread T si ricorre ad S

La funzione di SWI7CH è tipicamente codificata in assembly. L'OVERHEAD del contest switch può aumentare molto in base all'occupazione della cache da parte del processo.

gli EVENTI ESTERNI

- INTERRUPT: segnali provenienti dall'HW che determinano un salto al Kernel
- TIMER: segnale periodico generato da un orologio HW ogni N ms

Le CPU prevedono una o più LINEE DI INTERRUZIONE, una o più LINEE DI INTERRUZIONE NON MASCHERABILI (NMI), un insieme di linee che formano l'identità dell'interrupt.

Il CONTROLLO PROGRAMMATOE delle interrupt (PIC) (formina: REGISTRO DELLE INTERRUZIONI)

- Innalza la priorità
- Riabilita il FF di abilitazione generale
- Salva i registri
- Passa il controllo all'handler vero e proprio (servizio, ISR)
< servizio dell'interrupt>
- Ripristina i registri
- Abbassa la richiesta dell'interrupt servito
- Disabilita il FF di abilitazione generale
- Ripristina il livello di priorità
- Esegue istruzione RTI

REGISTRO DI MASCHERA

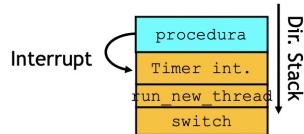
CIRCUITO DI PRIORITIZZAZIONE DELLE INTERRUZIONI

FF di ABILITAZIONE GENERALE delle interruzioni

TIMER (RTT o frequenza del clock)

* NB: il programmatore non ha mai controllo
ne garantisce nulla riguardo alla sequenza di esecuzione
del thread realizzato a BASSO LIVELLO

Commutazione determinata da timer



Per quanto riguarda le scelte del THREAD da mettere in esecuzione * il DISPATCHER può avere una delle seguenti situazioni:

1) NESSUN THREAD PRONTO → esegue l'OCHE THREAD ("rento in mano")

2) UN SOLO THREAD PRONTO → uso quello

+ PAIR, salvo il thread più in alto

3) PIÙ THREAD PRONTO → scelgo in base a criteri di priorità (LIFO, FIFO, nella coda con priorità)

- Il timer assicura istanti di decisione di scheduling
- Timer Interrupt routine:

```
TimerInterrupt() {  
    DoPeriodicHouseKeeping();  
    run_new_thread();  
}
```

- Risolve il problema di "controllare" l'esecuzione dei thread (preemption) → preemptive multithreading

DAI PROCESSI ALLA PROGRAMMAZIONE CONCORRENTE

In un intorno multiprogrammato la CPU esegue alternativamente SEQUENZE DI OPERAZIONI APPARTENENTI A PROCESSI DIVERSI. In un dato intervallo l'esecuzione di un processo può essere sospesa e ripresa più volte. Bisogna distinguere tra attivita della CPU e l'esecuzione di un particolare processo.

In un intorno multiprogrammato sono presenti contemporaneamente più processi/thread → se tu disponi di una sola CPU puoi eseguire un solo processo in qualsiasi momento (gli altri sono sospesi)

Lo STATO di un processo/thread

- IN ESECUZIONE → running
- BLOCCATO → idle, waiting
- PRONTO → ready

PROCESSO → esecuzione delle sequenze di operazioni molte per eseguire un determinato programma indipendente dalla CPU fisica e t-exe.

(o CPU può essere comandata in qualsiasi istante da un processo ad un altro → occorre prima salvare tutte le info. contenute nei registri della CPU e relativa al processo/thread che è stato sospeso)

L'**ALGORITMO** è un procedimento logico che deve essere eseguito per risolvere il problema in esame.

Il **PROGRAMMA** è la descrizione dell'algoritmo tramite un formalismo (linguaggio di programmazione) che rende possibile l'esecuzione dell'algoritmo da parte di un particolare elaboratore.

→ descrive più processi. (SARÀ PERTOGLIATA PARTE SUI GRAFI DI PRECEDENZA)

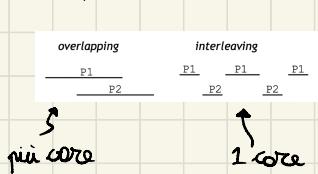
) linguaggi di programmazione non sequenziale consentono di descrivere una elaborazione non sequenziale come composizione di problemi sequenziali eseguiti contemporaneamente, ma analizzati e programmati separatamente

) problemi sequenziali concorrenti → **PROCESSI INDEPENDENTI**: grafo di precedenza è costituito da un insieme di rettangoli ad ORDINAMENTO ROTAZIONE e non connessi tra loro
→ **PROCESSI INTERAGENTI**: grafo di precedenza è un GRAFO CONNESSO ad ORDINAMENTO PARZIALE ($\{P_1, P_2\}$ su cui vale un ordinamento totale)

) processi indipendenti sono chiamati anche **PROCESSI ASINCRONI** → l'evoluzione di un processo non influenzava quello dell'altro

Per i processi interagenti bisogna rispettare i **VINCOLI DI PRECEDENZA** fra gli eventi e tra le operazioni (**VINCOLI DI SINCRONIZZAZIONE**)

Due processi/thread si dicono **CONCORRENTI** se la loro esecuzione si sovrappongono nel tempo.

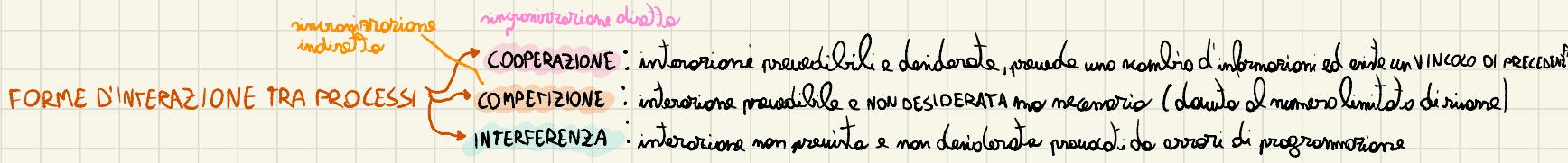


→ se la prima operazione di un processo/thread (inizio prima dell'ultima operazione dell'altro

Un'esecuzione in parallelo può essere implementata per combinare risorse fisiche, risorse logiche o per incrementare la velocità d'esecuzione (stesso t.t.o di esecuzione)

CARATTERISTICHE PROCESSI INDEPENDENTI: solo parallela non condizionata da altri processi, esecuzione deterministica (dipende solo dalla suddivisione di task in esecuzione riproducibile e può essere bloccata e fatto riportare senza molesto domani) (task in parallelo tra loro)

CARATTERISTICHE PROCESSI INTERAGENTI: solo condizionata da altri processi, esecuzione non deterministica (risultato non prevedibile) e non riproducibile



Eliminare le interferenze è un problema fondamentale della programmazione concorrente mondo tecnico di programmazione concorrente strutturata

PROCESSI CONCORRENTI E MUTUA ESCLUSIONE (x orale)

Le **INTERFERENZE** ponono enere di 2 tipi ↗ I TIPO: incremento di interazioni tra thread o processi non necessarie **SOL.** → incapsulamento dell'esecuzione
II TIPO: risoluzione erronea di interazioni tra thread o processi comunque necessarie **SOL.** ~~È malgrado di~~ del S.O.

	Ti	Tj
	contatore := contatore + 1 ...	contatore := contatore + 1 ...
	LOAD INCR STORE	A, contatore A, contatore, A in assembly
10: 11: 12: 13: 14: 15:	LOAD LOAD INCR STORE INCR STORE	A, contatore A, contatore A, contatore, A A, contatore, A
	(T1) (T1) (T1) (T1) (T1)	(T1) (T1) (T1) (T1) (T1)

possibile sequenza di esecuzione

~ In t_4 viene incrementato il contatore caricato da T_1 in t_0
ma viene ignorato l'incremento di T_2 in t_2

Si ha una **MUTUA ESCLUSIONE** quando non più di un processo alla volta può accedere ad un insieme di variabili comuni.

P1

P2

t

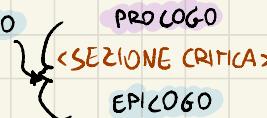
Nessun vincolo è imposto sull'ordine con il quale le operazioni sulle variabili comuni sono eseguite

La **SEZIONE CRITICA*** è la sequenza di istruzioni con le quali un processo accede e modifica un insieme di variabili comuni

La **REGOLA DI MUTUA ESCLUSIONE** afferma che: una sola sezione critica può essere in esecuzione in ogni istante

* **SCHEMA GENERALE**: ad ogni clone di sezione critica viene associato un **INDICATORE** {
 LIBERO se nessuna sezione critica è in esecuzione
 OCCUPATO se una sezione critica è in esecuzione

Ogni processo che vuole utilizzare una sezione critica della linea erge una sequenza di ordini, chiamata **PROTOCOLLO** che invoca una sequenza di ordini fra PROLOGO ed EPILOGO



Nello SCHEMA GENERALE, nelle fasi di PROLOGO si: a) modifica lo stato dell'indicatore: ne OCCUPATO \Rightarrow WAIT, strumenti 6)
b) modifica lo stato dell'indicatore

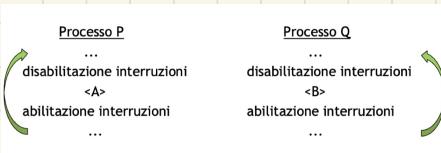
c) <esecuzione sezione critica>

EPILOGO d) modifica lo stato dell'indicatore (indicatore = libero)

Abbiamo diverse soluzioni per il problema della MUTUA ESCLUSIONE

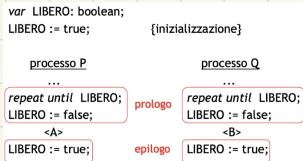
1^o sol: PROLOGO: disabilita interruzioni, EPILOGO abilita le interruzioni

! applicabile solo in sistemi con processore, elimina ogni possibilità di 1, il sistema è insensibile ad ogni tipo di interruzione



Da cui deriviamo il **1° REQUISITO** per la soluzione al problema: "Le versioni critiche devono essere eseguite con INTERRUZIONI ABILITATE"

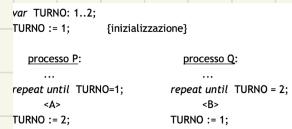
→ **2° SOL:**



→ Potrebbe accadere che P e Q romano eseguire <A> e CONTEMPORANEAMENTE, non va bene!

2° REQUISITO: Le versioni critiche di una stessa clona devono essere eseguite in forma MUTUAMENTE ESCLUSIVA

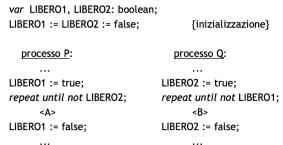
→ **3° SOL:**



→ un processo non può essere eseguito due volte di fila, i processi sono trattati in modo NON UNIFORME, dando priorità ad uno di essi. Le esecuzioni delle versioni critiche si alternano: se un processo si blocca prima della versione critica l'altro non può entrare.

3° REQUISITO: se un processo si blocca all'esterno di una versione critica, ciò non deve influenzare gli altri processi.

→ **4° SOL:**



→ se P e Q entrano contemporaneamente in REPEAT si può avere una condizione di tollo (o deadlock)

4° REQUISITO: La soluzione al problema della mutua esclusione deve essere priva di condizioni di tollo.

Per ragioni di efficienza nell'uso delle risorse è opportuno che un processo rilasci il controllo della CPU quando deve rimanere in attesa

→ **5° REQUISITO**: La soluzione non deve presentare fenomeni di ATTESA ATTIVA (BUSY WAITING)

→ **6° REQUISITO**: La soluzione non deve disabilitare per sempre uno o più processi (STARVATION)

PRIMI ALGORITMI PER RISOLVERE MUTUA ESCLUSIONE (xorole)

ALGORITMO DI

DEKKER (Dykstra)

il terzo turno tra 2 processi, quando un processo vuole entrare in res critico

importa LIBERO= false e importa il turno all'altro processo.

Successivamente entra in un ciclo di attesa finché l'altro processo non lo finisce

BU4 WAITING ~ estensione del caso N processi comporta complicazioni

```
var LIBERO1, LIBERO2: boolean;           TURNO: 1..2;
LIBERO1 := LIBERO2 := false; TURNO := 1; [inizializz.]
```

processo P:

```
...  
LIBERO1 := true;  
while LIBERO2 do  
if TURNO = 2 then  
begin  
LIBERO1 := false;  
repeat until TURNO = 1;  
LIBERO1 := true;  
end;  
<A>  
TURNO := 2;  
LIBERO1 := false;  
...
```

processo Q:

```
...  
LIBERO2 := true;  
while LIBERO1 do  
if TURNO = 1 then  
begin  
LIBERO2 := false;  
repeat until TURNO = 2;  
LIBERO2 := true;  
end;  
<B>  
TURNO := 1;  
LIBERO2 := false;  
...
```

RISPETTA:

2° REQ: i due processi non possono essere nello stesso tempo nella sezione critica

↳ Q modifica LIBERO2 mentre P LIBERO1

P entra in res. critica solo se LIBERO2 = FALSE e verifica LIBERO2 dopo aver posto LIBERO1 = false

3° REQ: se un solo processo chiede di entrare nella sezione critica il suo accesso è sempre garantito

4° REQ: nel corso di accessi simultanei la variabile turno consente di arbitrare l'accesso dei processi

6° REQ: Non c'è formazione, nel corso maggiore, ovvero se P è interrotto subito dopo il repeat until, il processo Q può guadagnare l'accesso solo per una volta ulteriore prima che entri P.

Nel ciclo while, ogni processo controlla il flag dell'altro processo e il turno corrente. Se il flag dell'altro processo è impostato a true e il turno è uguale all'altro processo, il processo corrente rimane bloccato nel ciclo while in attesa che l'altro processo finisca la sua sezione critica e ceda il turno.

Durante l'attesa attiva, il processo continua a verificare periodicamente le condizioni all'interno del ciclo while senza rilasciare la CPU o il tempo di esecuzione ad altri processi o thread. Questo approccio di attesa attiva può consumare risorse del sistema poiché i processi impegnano attivamente la CPU anche quando non possono fare progressi.

ALGORITMO DI PETERSON

```
var LIBERO1, LIBERO2: boolean; TURNO: 1..2;  
LIBERO1 := LIBERO2 := false; {inizializzazione}
```

processo P:

...

LIBERO1 := true;

TURNO := 2;

while LIBERO2

and TURNO = 2 do;

<A>

LIBERO1 := false;

...

processo Q:

...

LIBERO2 := true;

TURNO := 1;

while LIBERO1

and TURNO = 1 do;

LIBERO2 := false;

...

Nell'algoritmo di Peterson, quando un processo desidera accedere alla sezione critica, imposta il proprio flag a 'true' e il turno all'altro processo. Successivamente, il processo entra in un ciclo while in attesa finché l'altro processo ha impostato il proprio flag a 'true' e il turno è ancora dell'altro processo. In pratica, un processo si mette in attesa quando l'altro processo ha la precedenza per accedere alla sezione critica.

e richiede che i processi siano cooperativi.

PETERSON VS. DEKKER

Turno vs. attesa attiva: Nel caso dell'algoritmo di Peterson, viene utilizzato un meccanismo di turno per determinare quale processo ha il diritto di accedere alla sezione critica. Il processo corrente impone il proprio turno all'altro processo e successivamente entra in un ciclo while in attesa che l'altro processo finisca la sua sezione critica e ceda il turno. D'altra parte, l'algoritmo di Dekker utilizza un'attesa attiva o busy waiting, in cui i processi continuano a controllare attivamente le condizioni all'interno di un ciclo while fino a quando non possono accedere alla sezione critica.

RISPETTA:

2° REQ: i due processi non ponono enze simultaneamente nella sezione critica:
nolo Q modifica LIBERO2, nolo P modifica LIBERO1. Però in ser. critica nolo re LIBERO1=true e (LIBERO2=false || TURNO=1). Si no ragionamento per Q. Sia P de Q modifia le variabili TURNO nolo prima d'entrare nel while

3° REQ: indipendenza delle sezioni critiche, se un nolo processi richiede di entrare in ser. crit. è sempre garantito.

4° REQ: evitare di deadlock:

- LIB.1=FALSE, LIB.2=FALSE \Rightarrow processi fuori dalla ser. critica
- LIB.1=TRUE , LIB.2=FALSE \Rightarrow P nella sezione critica
- LIB.1=TRUE , LIB.2=TRUE, TURNO=1 \Rightarrow entra P
- no ragionamento per Q \Rightarrow V condizione di nolo non in lib

6° REQ: evitare di starvation: il processo in altro P_i potrebbe procedere non appena la sezione critica viene liberata da P_j (LIBERO_{-j}=false). Un eventuale tentativo di uscire da parte di P_j prima di P_i; rimane in coda perché P_j tiene TURNO=i

ALGORITMO DEL FORNAIO

Algoritmo di mutua esclusione con N processi (con busy waiting) ~ "MODELLO COME SERVIZI CLIENTI DI UN NEGOZIO"

↳ ciascun cliente (processo) riceve un numero quando entra nel negozio (sezione critica) viene rinnovato il cliente con il numero più basso (+ clienti non ricevono lo stesso numero)

- Variabili condivise tra gli N processi:

```
var CHOOSING: array [0 .. N - 1] of boolean initial false;  
NUMBER: array [0 .. N - 1] of integer initial 0;
```

```
repeat  
    ...  
    CHOOSING[i] := true; Eseguito da ciascun processo Pi che partecipa alla sezione critica  
    NUMBER[i] := i + max(NUMBER[0], ..., NUMBER[N-1]);  
    CHOOSING[i] := false;  
    for j := 0 to N-1  
        do begin  
            while CHOOSING[j] do;  
            while NUMBER[i] <> 0  
                and (NUMBER[j], j) < (NUMBER[i], i) do;  
        end;  
    <A>  
    NUMBER[i] := 0;  
    ...  
forever;
```

~ "va bene per la COORDINAZIONE DISTRIBUITA
il processo non cosa fanno gli altri processi"

Il PROLOGO è costituito da 2 fasi: acquisizione numero d'ordine e turno del turno di servizio

Se più clienti ricevono lo stesso numero → seguono ordine alfabetico (SCELTA DETERMINISTICA)

RISPETTA:
1° REQ: interrupt diritti

2° REQ: mutua esclusione: se due clienti P_i e P_k si trovano nel negozio e P_i è entrato nel negozio prima che P_k acquisire il numero NUM[i] < NUM[k].

Se P_i in SEZIONE CRITICA & P_k è nel negozio → (NUM[i], i) < (NUM[k], k) → AC PIÙ UN PROCESSO PUÒ TROVARSI IN SEZIONE CRITICA

3° REQ: indipendenza agli algoritmi unicamente soddisfatta
senza condizioni di deadlock

4° REQ: l'ordinamento totale ($num[i], i$) significa che solo uno dei processi in attesa può entrare nella sezione critica quando non è libera

6° REQ: Assenza di Starvation, i processi vengono serviti in ordine First Come First Served ed il numero massimo di turni di attesa di un processo è $N-1$

L'ALGORITMO DEL FORMAIO è stato concepito per essere usato in un ambiente decentralizzato (ogni processo dispone di una memoria locale) gli array CHOOSING e NUMBER sono distribuiti su diverse memorie, in caso di guasto di un nodo gli altri processi NON vengono bloccati → GRACEFUL DEGRADATION

CONTRO: Busy waiting, necessitano un numero finito e noto di processi/thread che devono concorrere e coordinarsi reciprocamente.

Mutua esclusione: Riepilogo dei requisiti

1. sezioni critiche eseguite con interruzioni abilitate
2. mutua esclusione dei thread nelle sezioni critiche
3. indipendenza dei thread rispetto alle sezioni critiche
4. assenza di condizioni di deadlock
5. assenza di attese attive
6. assenza di starvation

- 3°, 5°, 6° ORDINE COGNITIVO, relativa alla correttezza dell'implementazione
- 1° e 2° ORDINE REALIZZATIVO, riguardano L'EFFICIENZA della sua implementazione
- 4° ORDINE REALIZZATIVO, relativa alla correttezza dell'implementazione

Nell'HP di SEZIONI CRITICHE BREVI si ponono brevemente i REQ 1 e 5 per la soluzione al problema della mutua esclusione

Quando un Thread è in una sezione critica non ha accesso ESCLUSIVO a dati condivisi modificabili e tutti i Thread dovranno chiedendo quei dati vengono mantenuti in altre.

- Il Thread non può bloccare mentre è in sezione critica e la codifica deve essere efficiente (es. evitare cicli infiniti)
- Se per un motivo il Thread termina → SO deve rilasciare il corrispondente vincolo di mutua esclusione

In corso di SEZIONI CRITICHE TUTTE BREVI

→ CASO MONOPROCESSORE: PROLOGO distribuzione delle interruzioni, EPILOGO distribuzione delle interruzioni

→ CASO MULTIPROCESSORE: realizzazione di prologo ed epilogo mediante **lock(x)** ed **unlock(x)**

lock(x): begin
repeat until $x = 1$;
 $x := 0$;
end;

unlock(x): begin
 $x := 1$;
end;

X=1 una sezione critica in esecuzione
X=0 una sezione critica in esecuzione

lock(x) ed **unlock(x)** devono essere INDIVISIBILI (ATOMICHE)

La mutua esclusione nell'esecuzione delle sezioni critiche appartenenti alla stessa classe si ottiene come:

Ti	Tj
...	...
lock(x)	lock(x)
<sezione critica>	<sezione critica>
unlock(x)	unlock(x)
...	...

Req 2 → OK, mutua esclusione garantita poiché può entrare un solo Thread in SEZ. CNT.

Req 3 → su X possono operare solo lock ed unlock eseguite unicamente dal thread che vogliono accedere alla sezione critica

Req 4 → siccome lock ed unlock sono primitive non possono innescare condizioni di deadlock

~ implica BUSY WAITING, se la sezione critica vale 0 => lock(x) continua a ciclare è accettabile perché per HP sono brevi

Req 5 \Rightarrow nello lock è presente un'altra attesa

Req 6 \Rightarrow la formazione è teoricamente possibile (dipendendo dalle condizioni realizzative)

Per rendere corretta la soluzione lock-unlock è necessario che lo lock sia INVISIBLE (numero primario)

\hookrightarrow normalmente l'HW supporta solo UNLOCK() come funzione INVISIBLE, è necessario introdurre altre istruzioni molto specifiche \hookrightarrow TEST_AND_SET() "due azioni consecutive senza interruzione"
 \hookrightarrow XCHG per ricombinare i contenuti di un registro e di una locazione di memoria in un "unico" registro.

```
lock(x):  
    begin  
        repeat until not test_and_set(x);  
    end
```



```
lock(x):  
    begin  
        priv := true;  
        repeat XCHG(x,priv);  
        until priv = false;  
    end
```

TTSCL

```
lock(x):  
    begin  
        while;  
        repeat until not test_and_set(x);  
    end
```

code locali a cosa non sono invalidate
 \Rightarrow inizialmente dello spettacolo di scrittura dei thread da tentare di evitare quando il lock è già stato acquisito da altri thread

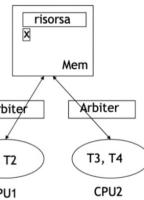
\hookrightarrow **CONTRO:** busy waiting dovuto allo spinning
soluzione valida solo per thread / processi residenti su CPU diverse

Il processore di Tj nell'interrogare x accede alla memoria e disturba Ti
processore di Tj è impegnato in una attività inutile: «spinning considered harmful»!

MUTUA ESCLUSIONE PER SEZIONI CRITICHE BREVI: PROTOCOLO GENERALE

« Possibile protocollo di accesso è:

```
Lock(x): {  
    Disable;  
    repeat until not test-and-set(x);  
    Enable;  
}  
<sez crit>  
Unlock(x): {  
    Disable;  
    Free(x);  
    Enable;  
}  
  
Così funziona!
```



\Rightarrow nello lock() si può utilizzare test-and-set con sequenza di istruzioni assembly

\Rightarrow adattabile a mono e multi-processore

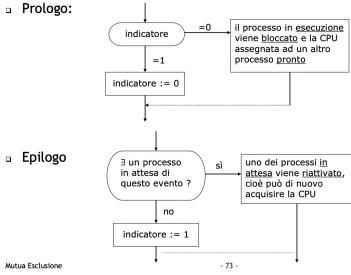
realizza RED LOGIC di mutua esclusione

non determina STARVATION (ma non lo prevede)

trascurare requisiti di efficienza: busy waiting

LIMITE + GRANDE

\Rightarrow applicabile solo a SEZ CRIT. BREVI

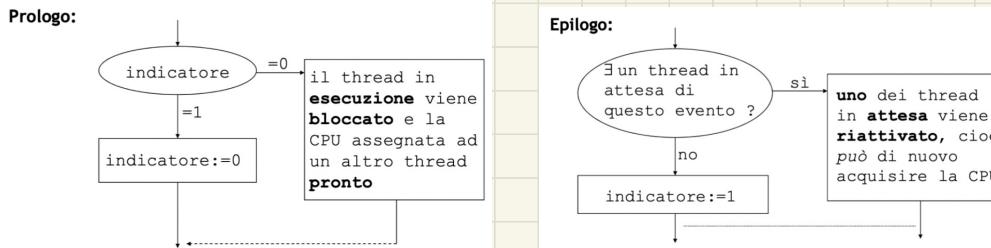


→ PROLOGO ed EPILOGO dovrebbero avocare operazioni atomiche "non dividibili"
 ↳ in ASSEMBLY richiedono molte operazioni
 ↳ occorre renderle ATOMICI

SEMAPORI E MUTUA ESCLUSIONE

L'idea controlla per **rendere più efficiente** anche i requisiti di EFFICIENZA: utilizzare il controllo di ritiro del processo Thread ed il meccanismo di esclusione degli istanti del thread offerto dal S.O.

→ OCCORRE: bloccare il thread per tutto il tempo in cui non ha accesso alla sezione critica (SOSPESO/WAITING)
 ↳ ristabilendo quando, per effetto del programma di altri thread, il suo accesso alla sezione critica è consentito (PRONTO oppure IN ESECUZIONE)
 ↳ ristabilire i thread in thread sulla sezione critica con una politica che non determini starvation



Un SEMAFORO è una variabile intera NON NEGATIVA ($s \geq 0$) con valore iniziale $s_0 \geq 0$. Al semaforo è associata una coda di attesa Q_s nella quale sono posti i descrittori dei processi che attendono l'autorizzazione a proseguire l'esecuzione.

Sul semaforo sono ormai solo 2 operazioni INDIVISIBILI (= PRIMITIVE): WAIT(n) ($P(n)$) e SIGNAL(n) ($V(n)$)
↳ realizzate tramite dinamica del s_0 ed eseguite in modo monitor → variabile semaforo è PROTETTA.

WAIT(n):

```
begin
  if  $s=0$  then {
    < sospendi thread e inserisci descrittore in  $Q_s$  >
  }
  else {
     $s := s - 1$ ;
  }
```



→ Può essere PASSANTE ($n > 0$) oppure BLOCCANTE ($n = 0$).

SIGNAL(n):

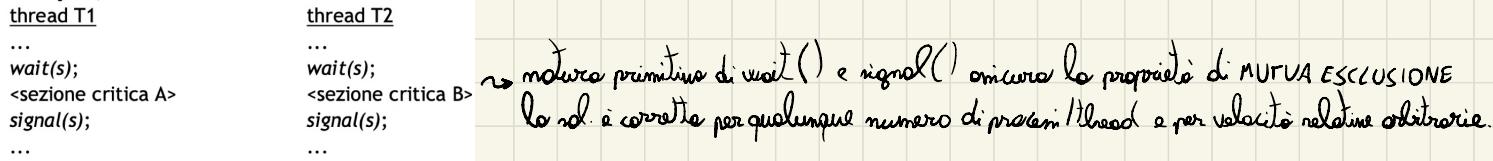
```
begin
  if < c'è un thread in coda  $Q_n$  > then {
    < il suo descrittore viene rimosso da  $Q_n$  e il suo stato modificato in pronto >
  }
  else {
     $n := n + 1$ ;
  }
end;
```



→ È sempre passante, non comporta comunque visualmente alcuna modifica allo stato del thread che l'ha eseguito. Dopo l'esecuzione di una signal(n) due thread sono potenzialmente in esecuzione.

Ogni classe di sezione critica viene associata una variabile `semaphore s`; protago ed epilogo sono realizzati tramite `wait()` e `signal()`

Supponiamo che A, B siano sezioni critiche della classe `classe`; `s` è un semaforo con valore iniziale `so=1`;



Occorre garantire che l'azione di andare a modificare del semaforo non sia separata dall'azione di sospensione.

per interi multipli occorre considerare `wait` e `signal` come SEZIONI CRITICHE BREVI e proteggerle mediante lock

signal(sem): begin
 <disabilitazione interruzioni>
 lock(x);
 <codice della signal>
 unlock(x);
 <abilitazione interruzioni>
end;

wait(sem): begin
 <disabilitazione interruzioni>
 lock(x);
 <codice della wait>
 unlock(x);
 <abilitazione interruzioni>
end;

LIVECCI SEZIONI CRITICHE → 1° LIV. sezioni critiche: S_1, S_2 , mutua esclusione: `wait` e `signal`
→ 2° LIV. sezioni critiche: `wait` e `signal`, mutua esclusione: `lock` ed `unlock`
→ 3° LIV. sezioni critiche: `lock(x)` ed `unlock(x)`, mutua esclusione: Tramite hardware (tek-ond-set) e/o disabilitazione interruzioni)

Poiché WAIT e SIGNAL sono le uniche operazioni ovunque sulla variabile semaforo, inizia:

$$val(s) = s_0 + ns(s) - nw(s)$$

$$\xrightarrow{val(s) \geq 0 \text{ sempre}} nw(s) \leq n_s(s) + s_0$$

$val(s) \rightsquigarrow$ valore semaforo

$s_0 \rightsquigarrow$ valore iniziale

$ns(s) \rightsquigarrow$ # di signal(s) eseguite

$nw(s) \rightsquigarrow$ # di wait(s) eseguite

↳ relazione invariante rispetto al numero
di primitive wait e signal eseguite.

- Un solo thread alla volta può trovarsi in sezione critica $\rightsquigarrow n = nw(s) - ns(s)$ // numero processi in sezione critica

(DIMOSTRAZIONE)

$$nw(s) \leq n_s(s) + s_0 \rightarrow nw(s) \leq n_s(s) + 1 \rightsquigarrow n = nw(s) - n_s(s) \leq 1 \text{ e poiché } nw(s) - n_s(s) \geq 0 \text{ ottiene } 0 \leq n \leq 1$$

- Un thread deve bloccarsi solo se la sezione critica è occupata

(DIMOSTRAZIONE)

$$\rightsquigarrow \text{se un thread viene bloccato } n=0 \xrightarrow{\text{relazione}} val(s) = n_s(s) - nw(s) + s_0 \rightsquigarrow nw(s) = n_s(s) + 1$$

↳ # wait supera il numero signal di 1, pertanto c'è un thread in sezione critica

La soluzione tramite WAIT e SIGNAL risolve in maniera generale il problema della mutua esclusione

R₁ OK perché le sezioni critiche possono essere eseguite con INTERRUZIONI ABILITATE, R₃ OK.

I meccanismi del SO per mutua esclusione (lock/unlock e/o semafori) sono variamente incapsulati da linguaggi e librerie, ... ovvero ...

Semafori e primitive di sincronizzazione sono realizzati dal linguaggio di programmazione o dalle librerie con funzionalità di base offerte dal SO

COOPERAZIONE TRA PROCESSI E THREAD MEDIANTE SEMAFORI

2 FORME DI COOPERAZIONE

SCAMBIO DI UN SEGNALE TEMPORALE che indica il verificarsi di un determinato evento.

SCAMBIO DI MESSAGGI generati da un thread e consumati da un altro.

- Prevede che l'esecuzione di alcuni di essi risulti condizionata dall'informazione prodotta da altri.
- Implica vincoli sull'ordinamento nel tempo delle operazioni dei thread.

Di seguito alcuni problemi di cooperazione di base risolti mediante semafori

SCAMBIO DI SEGNALI

N thread T_1, T_2, \dots, T_n attivati ad intervalli di tempo prefissi da un thread manager T_0

VINCOLI: l'esecuzione di T_i non può iniziare prima che sia giunto il segnale da T_0
ad ogni segnale inviato da T_0 deve corrispondere una attivazione da T_i

Ogni thread può essere regolato mediante un semaforo s_i con valore iniziale $s_{i,0} = 0$

```
thread  $T_i$ :  
begin  
  ...  
  repeat  
    wait(si);  
    ...  
    forever  
  end;  
  
thread  $T_0$ : //manager  
begin  
  ...  
  repeat  
    ...  
    signal(si);  
    ...  
    forever  
  end;
```

VERIFICA CORRETTEZZA

$m_i = \#$ richieste di attivazione da parte di T_i

$n_2 = \#$ segnali di attivazione inviati da T_0

$n_3 = \#$ volte in cui T_i è stato attivato

$nwl(n) \leq m_i(n) + n_0$ durante

$nwl(n) \leq m_i(n_i)$

$\Rightarrow n_2 \geq n_1 \Rightarrow n_3 = n_1$ in ogni istante
 $\Rightarrow n_2 < n_1 \Rightarrow n_3 = n_2$

PRODUTTORE - CONSUMATORE

HP: buffer in grado di contenere N messaggi, a cui accedono il processo P in scritto ed il processo C in letto

VINCOLI: il produttore non può inserire un messaggio nel buffer se questo è pieno
il consumatore non può prelevare un messaggio dal buffer se questo è vuoto

Vincolo correttivo: $0 \leq d - e \leq N$ ove
d = numero dei messaggi depositati
e = numero dei messaggi estraatti
N = dimensione del buffer

- Schema iniziale come processi asincroni:

<u>Produttore (P)</u>	<u>Consumatore (C)</u>
repeat	repeat
<produzione messaggio>	<prelievo messaggio>
<deposito messaggio>	<consumazione messaggio>
forever	forever

- La soluzione richiede due semafori:

'messaggio disponibile'	mess-disp v.i. = 0
'spazio disponibile'	spazio-disp v.i. = N

<u>Produttore (P)</u>	<u>Consumatore (C)</u>
repeat	repeat
<produzione messaggio>	wait(mess-disp)
wait(spazio-disp)	<prelievo messaggio>
<deposito messaggio>	signal(spazio-disp)
signal(mess-disp)	<consumazione messaggio>
forever	forever

→ produttore e consumatore non devono mai accedere alla stessa posizione del buffer

availability
redirection

BUFFER CIRCOLARE

Inserimento: *primo celle libere*
buffer(PUNT1):=messaggio_prodotto;
PUNT1:=(PUNT1+1) mod N;

Prelievo:
messaggio_prelevato:=buffer(PUNT2);
PUNT2:=(PUNT2+1) mod N;

* *ultimo celle rotte*

<u>Produttore</u> (Pi)	<u>Consumatore</u> (Cj)
<i>repeat</i>	<i>repeat</i>
<produzione messaggio>	<i>wait(mess-disp)</i>
<i>wait(spazio-disp)</i>	<i>wait(mutex2)</i>
<i>wait(mutex1)</i>	<prelievo messaggio>
<deposito messaggio>	<i>signal(mutex2)</i>
<i>signal(mutex1)</i>	<i>signal(spazio-disp)</i>
<i>signal(mess-disp)</i>	<consumazione messaggio>
<i>forever</i>	<i>forever</i>
□ mutex1, mutex2 v.i. 1; mess-disp v.i. 0; spazio-disp v.i. N	

MUTUA ESCLUSIONE CON INSIEME DI RISORSE EQUIVALENTI

- n risorse $\{R_1, \dots, R_n\}$ tra loro equivalenti
- m thread $\{T_1, \dots, T_m\}$ devono operare in modo ESCLUSIVO su una qualsiasi risorsa R_j , mediante una tra le operazioni $\{A, B, \dots\}$

- $R_j.A$ rappresenta l'operazione A eseguito su R_j

↳ POSSIBILE SOC. MEDIANTE SEMAFORI: ad ogni risorsa R_j viene assegnato un semaforo di mutua esclusione M_j con v.i. = 1

where iniziale

Thread Ti:

```

...
wait(Mj);
Rj.A;
signal(Mj);
...

```

- Come decide il generico thread Ti su quale risorsa R_j operare? come viene scelta la risorsa?
- Ti, una volta scelta la risorsa R_j , può rimanere bloccato eseguendo $wait(Mj)$ perché su R_j sta già operando un altro thread Tk. Ti si blocca su $wait(Mj)$ pur essendo disponibili altre risorse R_h ($h \neq j$)

SOLUZIONE

→ Introdurre una nuova risorsa G, gestore di $\{R_1, \dots, R_n\}$. G è costituito da una struttura dati destinata a mantenere lo stato delle risorse gestite. Si opera mediante RICHIESTA (vor x; i..n) o RILASCIO (x) rappresenta l'indice della risorsa assegnata o rilasciata, le due procedure devono essere eseguite in mutua esclusione → semaforo MUTEX (v.i.=1)

PROCEDURE

- Semafori: oltre al `mutex` (v.i. = 1) occorre un semaforo `ris` (v.i. = n) per indicare il *numero di risorse disponibili*
- Un vettore di variabili booleane `Libero[i]` registra quali risorse sono in ciascun istante libere (`Libero[i] = true`) e quali occupate (`Libero[i] = false`)

▫ thread T_i :

```

var   j: 1..n;
...
Richiesta(j); //attende su j indice della risorsa assegnata
<uso della risorsa j-ma>
Rilascio(j);
...

```

```

var   mutex: semaforo v.i.= 1; ris: semaforo v.i. = n;
      Libero: array [1..n] of Boolean v.i. =true;
procedure Richiesta(var x: 1..n);
var i: 0..n;
begin
  wait(mutex);
  wait(mutex);
  i := 0;
  repeat i := i + 1;
  until Libero [i];
  x := i;
  Libero [i] := false;
  signal(mutex);
end

procedure Rilascio(x: 1..n);
begin
  wait(mutex);
  Libero [x] := true;
  signal(mutex);
  signal(ris);
end

```

L'uso dei semafori in problemi di sincronizzazione più complessi solo male, occorre introdurre meccanismi di più alto livello

SEMAFORI PRIVATI

Finora la decisione per un processo di proseguire nell'esecuzione dipende da un solo semaforo.

mediante **primitiva signal(1)** (tipo FIFO)

Per problemi più complessi è necessario realizzare POLITICHE DI GESTIONE DELLE RISORSE (in modo implicito, al di fuori delle primitive di sincronizzazione)

- decisione circa la possibilità per un processo o un thread di proseguire nella esecuzione dipende dai verificarsi di una **CONDIZIONE DI SINCRONIZZAZIONE**
- lo scatto del processo da riattivare può dunque avvenire sulla base di condizioni di priorità tra i processi.

PRODUTTORI CONSUMATORI (MESSAGGI DI LUNGHEZZA VARIABILE)

O differenza della versione precedente, n'ha a disposizione un buffer di N celle cui poniamo di accedere più produttori e consumatori. I produttori inviano messaggi di lunghezza variabile l ($1 \leq l \leq N$)

→ E.S. DI POLITICA DI GESTIONE: dare priorità di accesso maggiore quello che fornisce il menaggio di maggior dimensione. (proviene STARVATION)

Per tutto il tempo in cui un produttore rimane sospeso con un messaggio di dimensione superiore allo spazio disponibile nel buffer, nessun altro produttore può depositare il proprio messaggio anche se le sue dimensioni possono essere contenute nello spazio libero del buffer

Condizione di sincronizzazione:

- "il deposito può avvenire se c'è sufficiente spazio per memorizzare e non ci sono produttori in attesa"

Al prelievo viene riattivato tra i processi sospesi quello il cui messaggio ha la dimensione maggiore, purché esista sufficiente spazio nel buffer. Se lo spazio disponibile non è sufficiente nessun produttore viene riattivato

Si definisce **SEMAFORO PRIVATO** per il processo P_i un semaforo priv. con $v.i = 0$, tale che: priv. è inizializzato a 0, solo P_i può eseguire la voce) (priv.) anche altri processi non possono eseguire la segnal.

- $SIGNAL(priv_i)$ \rightarrow altri processi
- $WAIT(priv_i)$ \rightarrow solo P_i

implementazione

per l'ACQUISIZIONE DELLA RISORSA da parte di un processo P_i

```
wait(mutex);  
<verifica della condizione di sincronizzazione  
    con esecuzione condizionata di signal(priv_i)>  
signal(mutex);  
wait(priv_i);  
<uso della risorsa>
```

se condizione di sincronizzazione è validata \Rightarrow modifica stato delle variabili eabilità acquisizione
risorsa tramite signal (priv_i)

RILASCIO DELLA RISORSA "

```
"  
wait(mutex);  
<verifica della condizione di sincronizzazione  
    con esecuzione condizionata di signal(priv_i)>  
signal(mutex);
```

Le non terminazione del processo chiude fuori dalla sezione critica per evitare deadlock

Se è presente una competizione tra alcuni processi per l'acquisizione della risorsa, i processi fanno uso di semafori distinti.

La soluzione completa sarà:

VARIABILI DI STATO: "sospesi" \Rightarrow numero processi PRODUTTORI sospesi

"vuote" \Rightarrow numero di celle vuote del buffer

"richiesta[i]" \Rightarrow numero di celle richieste dal produttore P_i non preso

PRODUTTORI: procedura di ACQUISIZIONE (include deposito nel buffer) \Rightarrow procedura acquisizione (l : integer, i : 1..num-prod)

```
procedure acquisizione(l: integer, i: 1..num-prod);  
begin  
    wait(mutex);  
    if sospesi = 0 and vuote >= l then  
        vuote := vuote - l;  
        signal(priv);  
    else  
        sospesi := sospesi + 1;  
        richiesta[i] := l;  
    end;  
    signal(mutex);  
end;
```

Condizione di sincronizzazione

Caratteristica del problema!
Condizione in base a cui il
thread può proseguire

se non ci sono processi in coda (è il mio turno!) ed ho spazio e sufficiente nel buffer
riduco memoria buffer
signal(priv);

Altrimenti, ci sono altri processi in coda \Rightarrow mi aggiungo alla coda

CONSUMATORI: procedura di RILASCIO del buffer (include politica rimozione dei processi sospesi) \Rightarrow rilascio(m: integer)

```
procedure rilascio(m: integer);  
begin  
    wait(mutex);  
    vuote := vuote + m;  
    exit := false;  
    while sospesi > 0 and not exit do  
        begin  
            <individuazione tra i processi sospesi del processo  $P_k$   
            con la massima richiesta, max>  
            if max <= vuote then  
                begin  
                    vuote := vuote - max;  
                    richiesta[k] := 0;  
                    sospesi := sospesi - 1;  
                    signal(priv);  
                end  
            else exit := true;  
        end;  
    signal(mutex);  
end;
```

Modello alternativo \Rightarrow ogni processo può usare una qualunque delle risorse (RISORSE EQUIVALENTI)

PROBLEMA DEI FILOSOFI CON SEMAFORI PRIVATI

LETTORE-SCRITTORI parla mutua esclusione

REGIONI CRITICHE

Dichiarazione: var v: shared T

↳ Uso della regione critica:

region V do $S_1, S_2, \dots S_n$ end

↳ regione critica, gli shared hanno accesso alla variabile shared v. Il compilatore può verificare che la variabile v sia mossa esclusivamente entro la regione critica e può realizzarne correttamente la mutua esclusione.

La regione critica semplice permette di realizzare solo malfatti di mutua esclusione.

```
var pila: shared record
    top: 0 .. N
    stack: array [0 .. N-1] of messaggio
begin top := 0 end /* valore iniziale */
procedure inserimento (y: messaggio);
region pila do
    if top = N then /* pila piena */
    else begin
        stack[top] := y;
        top := top + 1
    end
end
end
```

```
procedure prelievo (var x: messaggio);
region pila do
    if top = 0 then /* pila vuota */
    else begin
        top := top - 1;
        x := stack[top]
    end
end
end
```

Il compilatore realizza il contratto RC prima dichiarandolo: var V: shared T → mutex v.i.-1 e poi per le istruzioni in cui si usa la RC:
: region V do S end → Wait(mutex)
↔ Signal(mutex)] traduzione del compilatore

Il compilatore controlla solo che i processi accedono a V solo entro la regione critica, e può riconoscere situazioni di DEADLOCK POTENZIALE

La dichiarazione della **regione critica condizionale (RC)** è identico a quello della RC semplice

"Var v: shared T; ma l'uso combini: region V **when B** do S₁, S₂...S_n end;" → consente di ritardare il comportamento di una regione critica fino a quando non si verifica la condizione B

DEADLOCK

MONITOR (SCIDES 15X + 18) ~ FONDAMENTALI PER SCRITTO E PROGETTO

PROBLEMI SEMAFORI **SE** ↳ scarsa scalabilità delle soluzioni, problema di astrazione della programmazione concorrente: utilizzando i semafori il dettaglio della SINCRONIZZAZIONE è esperto in tutti i thread.

Il **MONITOR**  è un PARADIGMA DI PROGRAMMAZIONE CONCORRENTE (può essere fornito dal linguaggio di programmazione oppure essere realizzato con meccanismi di più basso livello)

PREVEDE

- SINCRONIZZAZIONE che riduzza la MUTUA ESCLUSIONE mediante semaforo mutex o un lock
- meccanismo per ritardare i thread: VARIABILE DI CONDIZIONE (possono essere diverse)

Un thread verifica la propria condizione logica di progresso DEFENDENDO il mutex e lo rilascia automaticamente se deve sospendersi.

```
type <nome del tipo> = monitor {  
    <dichiarazione di variabili locali>  
    procedure entry <nome1> ( ... );  
    begin ... end;  
    procedure entry <nome2> ( ... );  
    begin ... end;  
    procedure <nome3> ( ... );  
    begin ... end;  
    procedure <nome4> ( ... );  
    begin ... end;  
  
    begin <statement iniziale> end;  
}
```

identifica definizione di un tipo di dato astratto

} procedure entry e non entry definiscono le operazioni che è possibile compiere

N.B. Le istanze di un monitor sono accessibili contemporaneamente da più thread

↳ le PROCEDURE ENTRY devono essere eseguite in MUTUA ESCLUSIONE, se linguaggio non supporta i monitor allora il programmatore dovrà inserire dei mutex block.

Lo scopo del monitor è di controllare l'assegnazione ad una risorsa a thread concorrenti in base a determinate politiche di gestione.

- ovviamente ci sono 2 livelli:
 1. un solo thread alla volta ha accesso al monitor
 2. controllo dell'ordine con cui i thread hanno accesso alla risorsa.

mediante

UNICO SEMAFORO MUTEX non protegge ogni procedura entry

del monitor

mediante

VARIABILI DI TIPO CONDIZIONE

VARIABILI DI TIPO CONDIZIONE: rappresenta una Coda nella quale saranno sospesi i thread. È realizzato mediante un campo valore ed un puntatore alla coda.

Il programmatore può prevedere tante variabili di condizione quante sono le CONDIZIONI LOGICHE DISTINTE per cui un thread può essere ritardato.

↳ + specifico, con limite: una condizione \vee thread (= semafori privati)

Le PROCEDURE del monitor agiscono sulle variabili di condizione, **cond.wait** e **cond.signal**

COND_WAIT → SOSPENDE SEMPRE IL THREAD*, E LIBERA IL MONITOR (rilasciando il mutex) * viene inserito nella coda associata alla variabile cond.

COND_SIGNAL → RENDE ATTIVO UN THREAD IN ATTESA, nel caso di coda vuota non ci sono effetti collaterali

N.B. nei MONITOR la wait è sempre BLOCCANTE, nei SEMAFORI può anche essere PASSANTE

↳ **LOCK**: il SEMAFORO MUTEX forniscono protezione ai dati condivisi, le VARIABILI DI CONDIZIONI sono code di thread in attesa entro il monitor

→ deve essere acquistato prima di accedere ai dati condivisi e rilasciato al termine

Anche dopo un'attesa in COND_WAIT, il thread deve ottenerne nuovamente il lock o il mutex per accedere.

ESEMPIO PRODUTTORE-CONSUMATORI

1. Definire un ADT (Tipo di dato astratto)

Tipo mailbox = monitor
("tipo")

```
type mailbox = monitor {
    var buffer: array [0 .. N-1] of messaggio; /* rappresenta oggetto */
    testa, coda: 0 .. N-1;
    cont: 0 .. N;
    non_pieno, non_vuoto: condition;           le due condizioni per attesa di P e C
procedure entry send (x: messaggio);
begin
    if cont = N then non_pieno.wait;          thread attende se cont=N
    buffer [coda] := x;
    coda := (coda + 1) mod N;
    cont := cont + 1;
    non_vuoto.signal ←                   risveglio di eventuale consumatore in attesa
end
procedure entry receive (var x: messaggio);
begin
    if cont = 0 then non_vuoto.wait;          il thread attende se buffer vuoto
    x := buffer [testa];
    testa := (testa + 1) mod N;
    cont := cont - 1;
    non_pieno.signal ←                   quando è risvegliato ci deve essere elemento
end
begin cont := 0; testa := 0; coda := 0 end      ha liberato risorsa, eventuale risveglio di produttore
} /* fine definizione del tipo */
```

N.B. per lo scritto bisogna sapere bene distinguere tipi con intenze;
i nomi per valore e per riferimento devono essere CHIARI

2. Dichiarazione delle variabili

Var a, b : messaggio;

Var B, D: mailbox;

"intenzio" 2 "tipo"

3. Uso del monitor B per sincronizzazione dei thread P e C

Produttore: B.send (a)

Consumatore: B.receive (b)

var a, b: messaggio;
B: mailbox;

Pi

...

B.send(a);

Ci

...

B.receive(b);

...

→ in questo caso non è possibile inserire e vedere in parallelo su posizioni diverse. Il compilatore introduce un unico mutex per tutte le procedure del monitor

REALIZZAZIONE DEL MONITOR TRAMITE SEMAFORI

Il compilatore genera ad ogni istanza di monitor:

- Un semaforo mutex inizializzato ad 1 per la mutua esclusione delle procedure del monitor.
- Un semaforo urgent inizializzato a 0 per effettuare la preemption dei thread regolati.
- Un contatore **urgencount** inizializzato a 0 per conteggio in ogni istante i thread regolati risparmi.
- Per ogni variabile COND di tipo condition un SEMAFORO **CONDSEM** inizializzato a 0 ed un contatore **CONDCOUNT** inizializzato a 0 \rightarrow x CONDSEM
COND COUNT

ed espande ogni procedura entry con un **PROLOGO** ed un **EPICOGO**

{
 ↳ `if(urgentcount > 0):`
 `wait (mutex);`
 `{`
 `if(urgentcount > 0):`
 `then signal(urgent);`
 `else signal(mutex);`
 `}`
 `}`
 `}`
---> "chiama prima il suo mutex il thread
che è dentro in urgent)"

Il compilatore si occupa anche di "espandere" le operazioni COND.WAIT e COND.SIGNAL (utilizzando le variabili di sistema **CONDSEM** e **COND COUNT**)

```
cond.wait:  
condcount := condcount + 1; /* tiene traccia della sospens. */  
if urgentcount > 0  
then signal(urgent); /* riattiva un thread preempted */  
else signal(mutex); /* o libera del monitor */  
wait (condsem); /* sospens. su semaforo condsem */  
condcount := condcount - 1;
```

```
cond.signal:  
urgentcount := urgentcount + 1;  
if condcount > 0 then  
begin  
  signal(condsem); /* se ci sono thread sospesi */  
  wait(urgent); /* risveglia */  
  /* sospensione thread segnalante */  
end  
urgentcount := urgentcount - 1;
```

* Il memoforo URGENT è introdotto per effettuare una preemption del regolatore (tronca "wait(urgent)" in cond.signal) per evitare che il thread regolatore modifichi la condizione prima che il thread regolato sia andato in esecuzione.

SEMANTICA HOARE \rightsquigarrow Signal and Wait, procedura per uscire dal monitor è: if urgentcount > 0 then signal(urgent); else signal(mutex)

SEMANTICA CONCURRENT-PASCAL \rightsquigarrow Signal and Exit: per comporre cond.signal() come ultima operazione delle procedure evitando di aggiungere memoforo urgent \rightsquigarrow prologo: wait(mutex); epilogo: signal(mutex)

cond.wait:	condcount := condcount + 1; signal(mutex); wait(condsem); condcount := condcount - 1;
cond.signal:	if condcount > 0 then signal(condsem); else signal(mutex); /* solo in questo caso il monitor viene liberato */

ESEMPI CENITORI-SCRITTORI / FILOSOFI ↳ GUARDA LE SCIDE

TIPI DI MONITOR

condizione che deve essere vera per garantire la correttezza del monitor

Ogni monitor raffigura e mantiene un INVARIANTE DI CORRETEZZA dello stato. Ogni volta che il monitor è liberato o comunitato l'invariante deve essere vero.

In tutti i monitor l'INVARIANTE DI BASE deve essere rispettato: prima di eseguire una cond.wait
prima di completare la procedura entry ed uscire dal monitor

- MONITOR IN SEMANTICA HOARE ↳ priorità di segnalato, verifica di condizione specifica tramite if (comporta + cond.wait)
- MONITOR IN SEMANTICA MESA ↳ segnalante mantiene il controllo del mutex del monitor dopo avere segnalato la variabile di condiz.

Le primitive (principali) che operano sulle variabili condizione sono denominate *wait* e *notify*

Notify significa che il thread bloccato sulla variabile condizione potrebbe poter proseguire! --> hint

L'invariante specifico del thread segnalato non è garantito

Il thread segnalato deve comunque valutare nuovamente la condizione, ed eventualmente ri-sospendersi (stile Mesa):

while not ok_to_proceed do cond.wait end;

anzichè (stile Hoare):

if not ok_to_proceed then cond.wait;